# Code Shape II
# Expressions & Assignment

COMP 412
Fall 2005

# Road Map for Class

First, look at code shape

- Consider implementations for several language constructs

Then, consider code generation

- Selection, scheduling, & allocation *(order dictated by Lab 3)*
- Look at modern algorithms & modern architectures
- Lab 3 will give you insight into scheduling
  - Solve a really hard problem
  - In Lab 1, allocation was over-simplified

If we have time, introduce optimization

- Eliminating redundant computations *(as with DAGs)*
- Data-flow analysis, maybe SSA-form

Start out with code shape for expressions …

# Generating Code for Expressions

The key code quality issue is holding values in registers

- When can a value be safely allocated to a register?
  - When only 1 name can reference its value
  - Pointers, parameters, aggregates & arrays all cause trouble
- When should a value be allocated to a register?
  - When it is both *safe* & *profitable*

Encoding this knowledge into the *IR*

- Use code shape to make it known to every later phase
- Assign a virtual register to anything that can go into one
- Load or store the others at each reference
- ILOC has textual "memory tags" on loads, stores, & calls
- ILOC has a hierarchy of loads & stores (*see the digression*)

Relies on a strong register allocator

# Generating Code for Expressions

```
expr(node) {
  int result, t1, t2;
  switch (type(node)) {
      case ×,÷,+,− :
          t1← expr(left child(node));
          t2← expr(right child(node));
          result ← NextRegister();
          emit (op(node), t1, t2, result);
          break;
      case IDENTIFIER:
          t1← base(node);
          t2← offset(node);
          result ← NextRegister();
          emit (loadAO, t1, t2, result);
          break;
      case NUMBER:
          result ← NextRegister();
          emit (loadI, val(node), none, result);
          break;
      }
      return result;
}
```

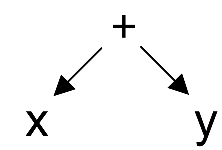
### The Concept

- Assume an AST as input & ILOC as output
- Use a postorder treewalk evaluator (*visitor pattern* in OOD)
  - Visits & evaluates children
  - Emits code for the op itself
  - Returns register with result
- Bury complexity of addressing names in routines that it calls
  - *base*(), *offset*(), & *val*()
- Works for simple expressions
- Easily extended to other operators
- Does not handle control flow

# Generating Code for Expressions

```
expr(node) {
  int result, t1, t2;
  switch (type(node)) {
       case ×,÷,+,– :
            t1← expr(left child(node));
            t2← expr(right child(node));
            result ← NextRegister();
            emit (op(node), t1, t2, result);
            break;
       case IDENTIFIER:
            t1← base(node);
            t2← offset(node);
            result ← NextRegister();
            emit (loadAO, t1, t2, result);
            break;
       case NUMBER:
            result ← NextRegister();
            emit (loadI, val(node), none, result);
            break;
        }
        return result;
}
```

Example:




Produces:

***expr("x") →***

| | | |
|---|---|---|
| loadI | @x | ⇒ r1 |
| loadAO | $r_{ARP}$, r1 | ⇒ r2 |

***expr("y") →***

| | | |
|---|---|---|
| loadI | @y | ⇒ r3 |
| loadAO | $r_{ARP}$, r3 | ⇒ r4 |

***NextRegister() →*** r5

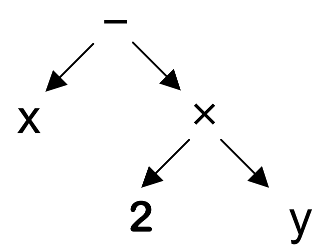
***emit(add,r2,r4,r5) →***

| | | |
|---|---|---|
| add | r2, r4 | ⇒ r5 |

# Generating Code for Expressions

```
expr(node) {
  int result, t1, t2;
  switch (type(node)) {
      case ×,÷,+,– :
          t1← expr(left child(node));
          t2← expr(right child(node));
          result ← NextRegister();
          emit (op(node), t1, t2, result);
          break;
      case IDENTIFIER:
          t1← base(node);
          t2← offset(node);
          result ← NextRegister();
          emit (loadAO, t1, t2, result);
          break;
      case NUMBER:
          result ← NextRegister();
          emit (loadI, val(node), none, result);
          break;
       }
       return result;
 }
```

**Example:**




**Generates:**

| | | | |
|---|---|---|---|
| loadI | @x | ⇒ | r1 |
| loadAO | $r_{ARP}$, r1 | ⇒ | r2 |
| loadI | 2 | ⇒ | r3 |
| loadI | @y | ⇒ | r4 |
| loadAO | $r_{ARP}$,r4 | ⇒ | r5 |
| mult | r3, r5 | ⇒ | r6 |
| sub | r2, r6 | ⇒ | r7 |

# Extending the Simple Treewalk Algorithm

More complex cases for IDENTIFIER

- What about values that reside in registers?
  - Modify the IDENTIFIER case
  - Already in a register ⇒ return the register name
  - Not in a register ⇒ load it as before, but record the fact
  - Choose names to avoid creating false dependences
- What about parameter values?
  - Many linkages pass the first several values in registers
  - Call-by-value ⇒ just a local variable with a negative offset
  - Call-by-reference ⇒ negative offset, extra indirection
- What about function calls in expressions?
  - Generate the calling sequence & load the return value
  - Severely limits compiler's ability to reorder operations

# Extending the Simple Treewalk Algorithm

Adding other operators

- Evaluate the operands, then perform the operation
- Complex operations may turn into library calls
- Handle assignment as an operator

Mixed-type expressions

- Insert conversions as needed from conversion table
- Most languages have symmetric & rational conversion tables

Typical Table for Addition

| + | Integer | Real | Double | Complex |
|---|---|---|---|---|
| Integer | Integer | Real | Double | Complex |
| Real | Real | Real | Double | Complex |
| Double | Double | Double | Double | Complex |
| Complex | Complex | Complex | Complex | Complex |

# Extending the Simple Treewalk Algorithm

What about evaluation order?

- Can use commutativity & associativity to improve code
- This problem is truly hard

Local rather than global

Commuting operands at one operation is much easier

- $1^{st}$ operand must be preserved while $2^{nd}$ is evaluated
- Takes an extra register for $2^{nd}$ operand
- Should evaluate more demanding operand expression first

(Ershov in the 1950's, Sethi in the 1970's)

Taken to its logical conclusion, this creates Sethi-Ullman scheme for register allocation [301 in EaC]

# Generating Code in the Parser

Need to generate an initial IR form

- Chapter 4 talks about ASTs & ILOC
- Might generate an AST, use it for some high-level, near-source work such as type checking and optimization, then traverse it and emit a lower-level IR similar to ILOC for further optimization and code generation

The Big Picture

- Recursive algorithm really works bottom-up
  - Actions on non-leaves occur after children are done
- Can encode same basic structure into *ad-hoc* SDT scheme
  - Identifiers load themselves & stack virtual register name
  - Operators emit appropriate code & stack resulting VR name
  - Assignment requires evaluation to an *lvalue* or an *rvalue*

# Ad-hoc SDT versus a Recursive Treewalk

```
expr(node) {
  int result, t1, t2;
  switch (type(node)) {
      case ×,÷,+,– :
          t1← expr(left child(node));
          t2← expr(right child(node));
          result ← NextRegister();
          emit (op(node), t1, t2, result);
          break;
      case IDENTIFIER:
          t1← base(node);
          t2← offset(node);
          result ← NextRegister();
          emit (loadAO, t1, t2, result);
          break;
      case NUMBER:
          result ← NextRegister();
          emit (loadI, val(node), none, result);
          break;
       }
       return result;
}
```

```
Goal :     Expr  { $$ = $1; } ;
Expr:      Expr PLUS Term
           { t = NextRegister();
             emit(add,$1,$3,t); $$ = t; }
        |  Expr MINUS Term  {…}
        |  Term { $$ = $1; } ;
Term:      Term TIMES Factor
           { t = NextRegister();
             emit(mult,$1,$3,t); $$ = t; };
        |  Term DIVIDES Factor {…}
        |  Factor { $$ = $1; };
Factor:    NUMBER
           { t = NextRegister();
             emit(loadI,val($1),none, t );
             $$ = t; }
        |  ID
            { t1 = base($1);
              t2 = offset($1);
              t = NextRegister();
             emit(loadAO,t1,t2,t);
             $$ =  t; }
```

# Handling Assignment (just another operator)

*lhs* ← *rhs*

Strategy

- Evaluate *rhs* to a value (*an rvalue*)
- Evaluate *lhs* to a location (*an lvalue*)
  - *lvalue* is a register ⇒ move rhs
  - *lvalue* is an address ⇒ store rhs
- If *rvalue* & *lvalue* have different types
  - Evaluate *rvalue* to its "*natural*" type
  - Convert that value to the type of **lvalue*

Unambiguous scalars go into registers

Ambiguous scalars or aggregates go into memory

Let hardware sort out the addresses !

# Handling Assignment

What if the compiler cannot determine the rhs's type ?

- This is a property of the language & the specific program
- If type-safety is desired, compiler must insert a run-time check
- Add a *tag* field to the data items to hold type information

Code for assignment becomes more complex

```
evaluate rhs
if type(lhs) ≠ rhs.tag
   then
      convert rhs to type(lhs)
or
      signal a run-time error
lhs ← rhs
```

This is much more complex than if it knew the types

# Handling Assignment

Compile-time type-checking

- Goal is to eliminate both the check & the tag
- Determine, at compile time, the type of each subexpression
- Use compile-time types to determine if a run-time check is needed

Optimization strategy

- If compiler knows the type, move the check to compile-time
- Unless tags are needed for garbage collection, eliminate them
- If check is needed, try to overlap it with other computation

Can design the language so all checks are static

# Handling Assignment (with reference counting)

The problem with reference counting

- Must adjust the count on each pointer assignment
- Overhead is significant, relative to assignment

Code for assignment becomes

```
evaluate rhs
lhs→count ← lhs→count - 1
lhs ← addr(rhs)
rhs→count ← rhs→count + 1
if (rhs→count = 0)
   free rhs
```

This adds *1 +, 1 -, 2 loads, & 2 stores*

With extra functional units & large caches, the overhead may become either cheap or free …